Capítulo QUATRO

Interfaces

Objetivos do Exame

Desenvolver código que declare, implemente e/ou estenda interfaces e use a anotação @Override.

O que é uma interface?

Na primeira vez que você olha para uma interface, você provavelmente vai pensar que ela é como uma classe com apenas definições de métodos:

```java
interface Monitorable {
    void monitor();
}
```

E, em termos práticos, você está certo.

Uma interface é um tipo de dado que apenas define métodos (abstratos) que uma classe deve implementar.

Embora, conceitualmente, seja mais interessante do que isso, porque isso permite que você defina o que uma classe pode fazer sem dizer como fazer. É por isso que se diz que uma interface é um contrato.

Qualquer classe que implemente uma interface deve fornecer uma implementação para todos os métodos da interface, caso contrário, a classe deve ser marcada como abstract.

Assim como uma classe é definida com a palavra-chave class, uma interface é definida com a palavra-chave interface.

Se uma classe quiser implementar uma interface, ela deve especificá-la com a palavra-chave implements.

Definindo uma interface

```java
public interface Monitorable {
    public static final int ID = 0;
    public abstract void monitor();
}
```

Implementando uma interface

```java
class Server implements Monitorable {
    public void monitor() {
        // Código da implementação
    }
}
```

Assim como uma classe, uma interface tem acessibilidade public ou padrão (default):

```java
public interface PublicAccessInterface {
    // ...
}

interface DefaultAccessInterface {
    // ...
}
```

Interfaces são abstratas por padrão (você não precisa especificar isso):

```java
public abstract interface PublicAccessInterface {
    // Isto é o mesmo que a definição acima
}
```

Isso significa duas coisas:

- Você não pode instanciar uma interface diretamente, ela deve ser implementada por uma classe para ser usada.
- Uma interface **não pode** ser marcada como final.

Os métodos definidos em uma interface são por padrão PUBLIC e ABSTRACT, o compilador os tratará como tais mesmo que você não os especifique.

Então, mesmo que você defina uma interface assim:

```java
interface Monitorable {
    void monitor();
    void setup();
}
```

Para o compilador, a interface será assim:

```java
interface Monitorable {
    public abstract void monitor();
    public abstract void setup();
}
```

Essa é a razão pela qual você **deve marcar o método como public** quando o implementa:

```java
class Server implements Monitorable {
    public void monitor() {
        // Implementação
    }
    public void setup() {
        // Implementação
    }
}
```

Campos declarados em uma interface são por padrão PUBLIC, STATIC e FINAL. Assim como os métodos, o compilador os tratará como tais mesmo que você não os especifique.

Isso significa que os campos são **CONSTANTES** ao invés de **VARIÁVEIS**:

```java
interface Monitorable {
    int ID = 0; // Você deve atribuir um valor no momento da criação
}
class Resource implements Monitorable {
    void change() {
        ID = 5; // ISTO NÃO compila
    }
}
```

Isso também significa que as seguintes declarações são todas equivalentes dentro de uma interface:

```java
int ID = 0;
public int ID = 0;
static int ID = 0;
final int ID = 0;
public static ID = 0;
public final ID = 0;
static final ID = 0;
public static final ID = 0;
```

Portanto, cuidado com declarações que **não vão compilar**, como estas:

```java
interface Monitorable {
    private int ID = 0; // Não pode ser private
    int TIMEOUT; // É final, então você precisa fornecer um valor
}
```

Há duas regras quanto à herança e interfaces:

1. Uma **classe** pode **implementar** (não estender) **qualquer número de interfaces**.
2. Uma **interface** pode **estender qualquer número de interfaces**, mas **não pode estender uma classe**.

Implementar uma interface é um tipo de herança.

Quando uma classe implementa uma interface, podemos usá-la assim para tirar proveito do **polimorfismo**:

```java
interface Monitorable {
    void monitor();
}
class Disk implements Monitorable {
    public void monitor() {
        System.out.println("Monitoring Disk");
    }
}
class Server implements Monitorable {
    public void monitor() {
        System.out.println("Monitoring Server");
    }
}
public class Test {
    public static void main(String args[]) {
        Monitorable m = new Disk();
        m.monitor();
        m = new Server(); // Troca a implementação
        m.monitor();
    }
}
```

A saída será:

```arduino
Monitoring Disk
Monitoring Server
```

Uma classe **não pode estender** mais de uma classe, mas **pode implementar** mais de uma interface. Você pode entender o motivo com um exemplo. Considere:

```java
class Truck {
    public void accelerate() {
        System.out.println("Accelerating truck...");
    }
}
class CompactCar {
    public void accelerate() {
        System.out.println("Accelerating compact car...");
    }
}
```

Se fosse possível herdar de múltiplas classes:

```java
class Car extends Truck, CompactCar {
    public void run() {
        accelerate();
        // ...
    }
}
```

Qual método accelerate() o Java escolheria?

Este é um problema que os projetistas do Java decidiram evitar **não permitindo herança múltipla**.

Mas e se estivermos usando interfaces?

```java
interface Truck {
    void accelerate();
}
interface CompactCar {
    void accelerate();
}
class Car implements Truck, CompactCar {
    public void accelerate() {
        System.out.println("Accelerating car");
        // ...
    }
}
```

Como interfaces **não fornecem uma implementação**, há apenas uma (a da classe Car), então **não há conflito** e o problema é evitado completamente!

E se os métodos tiverem o mesmo nome mas parâmetros diferentes?

```java
interface Truck {
    void accelerate();
}
interface CompactCar {
    void accelerate(int speed);
}
```

Eles são considerados **dois métodos diferentes** porque a assinatura do método é diferente. A classe implementadora terá que implementar **ambas** as versões:

```java
class Car implements Truck, CompactCar {
    public void accelerate() {
        // implementação
    }
    public void accelerate(int speed) {
        // implementação
    }
}
```

Contudo, quando os métodos só diferem no tipo de retorno, como o tipo de retorno **não faz parte da assinatura do método**, o compilador do Java gerará um erro:

```java
interface Truck {
    void accelerate();
}
interface CompactCar {
    int accelerate();
}
class Car implements Truck, CompactCar {
    // O Java reclamará de métodos duplicados
    // e tipos de retorno incompatíveis
    public void accelerate() {
        // implementação
    }
    public int accelerate() {
        // implementação
    }
}
```

Opcionalmente, para tornar as coisas mais claras, podemos usar a anotação @Override:

```java
interface CompactCar {
    int accelerate();
}
class Car implements CompactCar {
    @Override
    public int accelerate() {
        // implementação
    }
}
```

A anotação @Override indica que um método **sobrescreve** uma declaração de método em um supertipo, seja em uma interface ou em uma classe-pai.

Se o método anotado **não sobrescrever ou implementar corretamente** o método, o compilador gerará um erro.

Essa é a única função da anotação. É útil em casos como quando há um erro **não tão óbvio** causado por um erro de digitação:

```java
interface CompactCar {
    int accelerate();
}
class Car implements CompactCar {
    // O compilador marcará erro sobre o método
    // "acelerate" não sobrescrever ou implementar algo
    @Override
    public int acelerate() {
        // implementação
    }
}
```

Por fim, uma interface **pode estender apenas outras interfaces**.

```java
interface Monitorable {
    void monitor();
}
interface Pluggable {
    void plug();
}
interface Resource extends Monitorable, Pluggable {
    void printInfo();
}
```

Uma classe **não abstrata** que implementa Resource deve implementar **todos os métodos** das três interfaces:

```java
class Disk implements Resource {
    public void monitor() {
        // implementação
    }
    public void plug() {
        // implementação
    }
    public void printInfo() {
        // implementação
    }
}
```

O que há de novo no Java 8?

Suponha que temos uma interface como esta:

```java
interface Processable {
    void processInSequence();
}
```

E uma implementação:

```java
class Task implements Processable {
    public void processInSequence() {
        System.out.println("Processing in sequence");
    }
}
```

Sabemos que quando uma classe implementa uma interface, **a menos que a classe seja marcada como abstract**, ela deve implementar **TODOS** os métodos dessa interface.

Então se adicionarmos outro método a Processable, por exemplo:

```java
interface Processable {
    void processInSequence();
    void processInParallel();
}
```

Teremos que atualizar a classe para evitar um erro de compilação:

```java
class Task implements Processable {
    public void processInSequence() {
        System.out.println("Processing in sequence");
    }
    public void processInParallel() {
        System.out.println("Processing in parallel");
    }
}
```

Isso foi fácil. Mas pense no seguinte:

- E se tivermos centenas de classes implementando Processable?
- E se não pudermos atualizar ou não tivermos acesso ao código por algum motivo?
- E se o novo método **não for necessário** ou **não fizer sentido** para algumas implementações?

Esses são problemas reais, às vezes difíceis de resolver.

Contudo, o **Java 8** nos dá os **métodos default**. Não precisamos fornecer implementações para eles porque **são métodos não abstratos**.

Em outras palavras, interfaces agora permitem métodos **com CORPO**. E isso **não é tão simples quanto parece**.

Métodos default

A principal razão para adicionar métodos default às interfaces foi **permitir a evolução das interfaces**, adicionando nova funcionalidade e, ao mesmo tempo, **garantindo compatibilidade** com o código escrito em versões anteriores.

Existem dois efeitos colaterais dignos de nota:

1. Agora podemos projetar **métodos opcionais**. Podemos ter métodos com funcionalidade limitada ou padrão, e as classes que implementam as interfaces podem decidir se mantêm essa funcionalidade ou fornecem outra.
2. Podemos ter **métodos utilitários diretamente na interface**. Métodos que obtêm ou criam recursos, por exemplo, feitos apenas por conveniência e possivelmente implementados em termos de métodos não-default da interface.

Contudo, agora que interfaces podem fornecer comportamento, a diferença entre elas e **classes abstratas** não é muito clara em alguns casos. Ainda assim, há duas diferenças significativas:

- Uma **classe** pode estender **somente UMA** classe abstrata, mas pode **implementar MÚLTIPLAS interfaces**.
- Uma classe abstrata pode ter estado por meio de **variáveis de instância (campos)**. Uma **interface NÃO PODE**.

Definindo um método default

```java
interface Processable {
    void processInSequence();
    default void processInParallel() {
        /** A implementação padrão vai aqui */
    }
}
```

Uma interface pode ter **qualquer número** de métodos abstratos e métodos default.

Todos os métodos com a palavra-chave default devem ter um **corpo**.

Métodos default são **implicitamente públicos**, como qualquer outro método de uma interface.

Ao tornar processInParallel() um método default, a classe que o implementa **o herda automaticamente**. Veja o exemplo completo:

```java
interface Processable {
    void processInSequence();
    default void processInParallel() {
        System.out.println("Processing in parallel");
    }
}
public class Task implements Processable {
    public void processInSequence() {
        System.out.println("Processing in sequence");
    }
    public static void main(String args[]) {
        Task t = new Task();
        t.processInSequence();
        t.processInParallel(); // Isso compila perfeitamente
    }
}
```

A saída será:

```nginx
Processing in sequence
Processing in parallel
```

Este é o cenário mais simples, onde a classe implementadora **herda o método default**.

Antes de apresentar cenários mais complexos, vejamos **quais são as restrições** ao usar métodos default.

Restrições dos métodos default

- Métodos default **não podem ser final**.
  Se um método é final, não pode ser sobrescrito pelas classes implementadoras, o que contraria o objetivo principal dos métodos default.
- Métodos default **não podem ser synchronized**.
  Essa foi uma decisão deliberada dos projetistas da linguagem. Se um método fosse synchronized na interface, significaria que todas as classes que o implementassem herdariam esse comportamento. Mas essa decisão deve pertencer à implementação; **a interface não tem base razoável para definir a política de sincronização**.

- Métodos default são sempre public.
  Como qualquer outro método de uma interface. Diferente de uma classe abstrata, onde você pode escolher a visibilidade.
- Você **não pode ter métodos default para métodos da classe Object**.
  Uma interface **não pode fornecer implementações default para**:

```java
boolean equals(Object o)
int hashCode()
String toString()
```

Se uma interface contiver métodos com essas assinaturas, o **compilador lançará um erro**.
O motivo é que esses métodos dizem respeito ao **estado do objeto**. Como **interfaces não têm estado**, esses métodos devem estar nas **classes implementadoras**.

Classe sobrescreve método default

As **classes sempre VENCEM** as interfaces.
Se uma classe sobrescreve um método default, o **método da classe será o usado**. Por exemplo:

```java
interface Processable {
    void processInSequence();
    default void processInParallel() {
        System.out.println("Default parallel");
    }
}
public class Task implements Processable {
    public void processInSequence() {
        System.out.println("Processing in sequence");
    }
    public void processInParallel() {
        System.out.println("Class parallel");
    }
    public static void main(String args[]) {
        Task t = new Task();
        t.processInParallel();
    }
}
```

A saída será:

```vbnet
Class parallel
```

Isso é verdade **mesmo que a classe redefina o método default como abstract**:

```java
interface Processable {
    void processInSequence();
    default void processInParallel() {
        System.out.println("Default parallel");
    }
}

// A classe Task precisa ser abstrata
abstract class Task implements Processable {
    public void processInSequence() {
        System.out.println("Processing in sequence");
    }
    public abstract void processInParallel();
}
```

Se por algum motivo, você precisar chamar a **implementação default** do método, pode fazer isso com o **nome da interface seguido de super**:

```java
public void processInParallel() {
    Processable.super.processInParallel();
}
```

Isso **só funciona com métodos default**.
Tentar chamar um método **não-default** dessa forma resulta em erro de compilação.
Além disso, super deve ser usado com uma interface **diretamente herdada** pela classe.

Outro cenário relacionado a essa regra é quando **um método herdado de uma classe** sobrescreve um **método default** de uma interface:

```java
interface Processable {
    default void processInParallel() {
        System.out.println("Default parallel");
    }
}
class Process {
    public void processInParallel() {
        System.out.println("Class parallel");
    }
}
public class Task extends Process implements Processable {
    public static void main(String args[]) {
        Task t = new Task();
        t.processInParallel();
    }
}
```

A saída é:

```vbnet
Class parallel
```

O método processInParallel() retorna a string "Class parallel" porque a classe Task herda esse método da classe Process, que **sobrescreve** o método default de mesmo nome da interface Processable.

Herança de interface com métodos default

Interfaces mais específicas sempre VENCEM interfaces menos específicas.

Os métodos default das interfaces **mais profundas** na hierarquia de herança serão usados. Por exemplo:

```java
interface Processable {
    void processInSequence();
    default void processInParallel() {
        System.out.println("Processable parallel");
    }
}
interface Parallelizable extends Processable {
    default void processInParallel() {
        System.out.println("Parallelizable parallel");
    }
}
public class Task implements Parallelizable {
    public void processInSequence() {
        System.out.println("Processing in sequence");
    }
    public static void main(String args[]) {
        Task t = new Task();
        t.processInParallel();
    }
}
```

A saída será:

```nginx
Parallelizable parallel
```

A interface Parallelizable herda o método default processInParallel(), mas como ela **o redefine**, é a sua versão que será usada pela classe Task.

Se Parallelizable definisse processInParallel() como abstract:

```java
interface Parallelizable extends Processable {
    abstract void processInParallel();
}
```

Então Task **teria que implementar o método**, para não ser uma classe abstrata:

```java
public class Task implements Parallelizable {
    public void processInSequence() {
        System.out.println("Processing in sequence");
    }
    public void processInParallel() {
        System.out.println("Task parallelizable");
    }
    public static void main(String args[]) {
        Task t = new Task();
        t.processInParallel();
    }
}
```

A saída seria:

```arduino
Task parallelizable
```

Herança múltipla de interfaces com métodos default

As classes podem implementar **múltiplas interfaces**.
O que acontece quando **duas interfaces têm o mesmo método default**?
Qual delas a classe implementadora escolhe?

Considere o exemplo:

```java
interface Processable {
    void processInSequence();
    default void processInParallel() {
        System.out.println("Processable parallel");
    }
}
interface Parallelizable {
    default void processInParallel() {
        System.out.println("Parallelizable parallel");
    }
}
public class Task implements Processable, Parallelizable {
    public void processInSequence() {
        System.out.println("Processing in sequence");
    }
    public static void main(String args[]) {
        Task t = new Task();
        t.processInParallel();
    }
}
```

O resultado será:

Erro de compilação:

```scss
Métodos default duplicados chamados processInParallel com os parâmetros () e ()
```

O compilador **não sabe qual versão escolher**, então gera um erro.

Neste caso, Task **deve fornecer sua própria implementação** (seguindo a regra de que a interface ou classe mais específica vence), sobrescrevendo os métodos default da interface e resolvendo o conflito:

```java
public class Task implements Processable, Parallelizable {
    public void processInSequence() {
        System.out.println("Processing in sequence");
    }
    public void processInParallel() {
        System.out.println("Task parallelizable");
    }
    public static void main(String args[]) {
        Task t = new Task();
        t.processInParallel();
    }
}
```

A saída agora será:

```
Task parallelizable
```

Claro, também podemos chamar uma implementação default diretamente com:

```java
public void processInParallel() {
    Processable.super.processInParallel();
}
```

Definindo um método static

```java
interface Processable {
    static void log() {
        /** Implementação vai aqui */
    }
}
```

Métodos static em interfaces são definidos **exatamente como em classes**, com a palavra-chave static.

- Métodos static são **implicitamente públicos**, como qualquer outro método da interface.
- Uma interface pode conter **qualquer número** de métodos static.

Métodos static

Sempre que nos referimos a algo static, estamos nos referindo a algo que pertence a uma **classe** (ou interface), e **não a uma instância ou objeto específico**.

Métodos static em interfaces seguem o mesmo conceito: **pertencem à interface onde são declarados**.

Eles foram adicionados para **ajudar os métodos default** e para **organizar melhor métodos utilitários**, porque geralmente esses métodos eram definidos em outra classe (como java.util.Collections), em vez de onde **naturalmente pertencem**.

Por exemplo, a interface java.util.Comparator define o método estático:

```java
static <T> Comparator<T> comparingInt(ToIntFunction<? super T> keyExtractor)
```

Usado por um método default:

```java
default Comparator<T> thenComparingInt(ToIntFunction<? super T> keyExtractor)
```

Métodos static de interface não são herdados.
Você deve prefixar o método com o nome da interface:

```java
interface Parallelizable {
    static void log(String s) {
        System.out.println(s);
    }
    default void processInParallel() {
        log("Parallelizable parallel");
    }
}
public class Task implements Parallelizable {
    public static void main(String args[]) {
        Task t = new Task();
        t.processInParallel();
        // t.log("The end"); Não compila
        // Task.log("The end"); Também não compila
        Parallelizable.log("The end"); // Compila!
    }
}
```

Saída:

```arduino
Parallelizable parallel
The end
```

Pontos-chave

- Uma **interface** é um tipo de dado que apenas define métodos (**abstratos**) que uma classe deve implementar.
- Uma interface é definida com a palavra-chave interface.
  Se uma classe quiser implementar uma interface, ela deve especificar isso com a palavra-chave implements.
- Uma interface tem acessibilidade public ou padrão (default) e é **abstrata por padrão**.
- Os métodos definidos em uma interface são, por padrão, public e abstract.
  O compilador os tratará assim mesmo que você não os declare explicitamente.
- Os campos declarados em uma interface são, por padrão, public, static e final.
  Assim como os métodos, o compilador os tratará assim mesmo que não sejam especificados.

Isso significa que os campos são **constantes**, não variáveis.

- Uma classe pode **implementar** (não estender) **qualquer número de interfaces**.
- Uma interface pode **estender qualquer número de interfaces**, mas **não pode estender uma classe**.
- A anotação @Override indica que um método sobrescreve uma declaração de método em um supertipo (interface ou classe-pai).

Se o método anotado não sobrescrever ou implementar corretamente o método, o compilador gerará um erro.

- O Java 8 introduziu **métodos default** em interfaces para suportar a evolução das interfaces, adicionando nova funcionalidade e, ao mesmo tempo, mantendo a compatibilidade com código legado.
- Métodos default são marcados com a palavra-chave default, e devem ter um corpo.
  As classes que os implementam podem usar ou sobrescrever esses métodos.
- Métodos default são sempre public, mas **não são static**.
  Eles **não podem ser synchronized nem final**.
- Você **não pode definir métodos default com a mesma assinatura dos métodos da classe Object**

```java
boolean equals(Object o);
int hashCode();
String toString();
```

- Em uma hierarquia de herança, o método **mais específico é o que será chamado**.
- Exemplo: se uma classe sobrescreve um método default, o método da classe será usado.
- Se uma interface sobrescreve um método default herdado de uma superinterface, o método da subinterface será usado.
- Se uma classe implementa **duas interfaces diferentes com o mesmo método default (mesma assinatura)**, ela **deve sobrescrever o método**. Caso contrário, ocorre um erro de compilação.
- Se você quiser chamar o método default de uma interface a partir da classe implementadora (ou de uma subinterface), faça assim:

```java
NomeDaInterface.super.metodoDefault();
```

- O Java 8 também introduziu métodos static em interfaces para que elas pudessem conter métodos utilitários.
- Métodos static são marcados com a palavra-chave static, e também devem ter corpo.
- Métodos static em interfaces têm o mesmo comportamento que os de classes: **não são herdados**.
- Se quiser chamar um método static de uma interface, faça assim:

```java
NomeDaInterface.metodoStatic();
```

Autoavaliação (Self Test)

---

1. Dado:

```java
interface A {
    default int aMethod() {
        return 0;
    }
}
public class Test implements A {
    public long aMethod() {
        return 1;
    }
    public static void main(String args[]) {
        Test t = new Test();
        System.out.println(t.aMethod());
    }
}
```

Qual é o resultado?

A. 0
B. 1
C. Falha de compilação
D. Uma exceção ocorre em tempo de execução

---

2. Dado:

```java
interface B {
    default static void test() {
        System.out.println("B test");
    }
}
public class Question_4_2 implements B {
    public void test() {
        System.out.println("Q test");
    }
    public static void main(String[] args) {
        Question_4_2 q = new Question_4_2();
        q.test();
    }
}
```

Qual é o resultado?

A. B test
B. Q test
C. Falha de compilação
D. Uma exceção ocorre em tempo de execução

---

3. Dado:

```java
interface C {
    default boolean equals(C obj) {
        return obj == this;
    }
}
public class Question_4_3 implements C {
    public static void main(String[] args) {
        Question_4_3 q = new Question_4_3();
        System.out.println(q.equals(q));
    }
}
```

Qual é o resultado?

A. true
B. false
C. Falha de compilação
D. Uma exceção ocorre em tempo de execução

---

4. Dado:

```java
interface D {
    default void print() {
        System.out.println("D");
    }
}
interface E extends D {
    default void print() {
        System.out.println("E");
    }
}
public class Question_4_4 implements E {
    public void print() {
        E.super.print();
    }
    public static void main(String[] args) {
        Question_4_4 q = new Question_4_4();
        q.print();
    }
}
```

Qual é o resultado?

A. D
B. E
C. D e depois E
D. Falha de compilação
E. Uma exceção ocorre em tempo de execução

---

5. Dado:

```java
interface F {
    static void test() {
        System.out.println("F test");
    }
}
public class Question_4_5 implements F {
    public void test() {
        System.out.println("Q test");
    }
    public static void main(String[] args) {
        F q = new Question_4_5();
        q.test();
    }
}
```

Qual é o resultado?

A. F test
B. Q test
C. Falha de compilação
D. Uma exceção ocorre em tempo de execução

---

6. Dado:

```java
interface G {
    default void doIt() {
        System.out.println("G - Do It");
    }
}
interface H {
    void doIt();
}
public class Question_4_6 implements G, H {
    public void doIt() {
        System.out.println("Do It");
    }
    public static void main(String[] args) {
        Question_4_6 q = new Question_4_6();
        q.doIt();
    }
}
```

Qual é o resultado?

A. G - Do It
B. Do It
C. Falha de compilação
D. Uma exceção ocorre em tempo de execução